Uncovering Bugs in Distributed Storage Systems during Testing (not in Production!)

Pantazis Deligiannis[†1], Matt McCutchen[◇1], Paul Thomson[†1], Shuo Chen[⋆]
Alastair F. Donaldson[†], John Erickson[⋆], Cheng Huang[⋆], Akash Lal[⋆]
Rashmi Mudduluru[⋆], Shaz Qadeer[⋆], Wolfram Schulte[⋆]

[†]*Imperial College London,* [◇]*Massachusetts Institute of Technology,* [⋆]*Microsoft*

Abstract

Testing distributed systems is challenging due to multiple sources of nondeterminism. Conventional testing techniques, such as unit, integration and stress testing, are ineffective in preventing serious but subtle bugs from reaching production. Formal techniques, such as TLA+, can only verify high-level specifications of systems at the level of logic-based models, and fall short of checking the actual executable code. In this paper, we present a new methodology for testing distributed systems. Our approach applies advanced systematic testing techniques to thoroughly check that the executable code adheres to its high-level specifications, which significantly improves coverage of important system behaviors.

Our methodology has been applied to three distributed storage systems in the Microsoft Azure cloud computing platform. In the process, numerous bugs were identified, reproduced, confirmed and fixed. These bugs required a subtle combination of concurrency and failures, making them extremely difficult to find with conventional testing techniques. An important advantage of our approach is that a bug is uncovered in a small setting and witnessed by a full system trace, which dramatically increases the productivity of debugging.

1 Introduction

Distributed systems are notoriously hard to design, implement and test [6, 31, 20, 27, 35]. This challenge is due to many sources of *nondeterminism* [7, 23, 32], such as unexpected node failures, the asynchronous interaction between system components, data losses due to unreliable communication channels, the use of multithreaded code to exploit multicore machines, and interaction with clients. All these sources of nondeterminism can easily create *Heisenbugs* [16, 38], corner-case bugs that are difficult to detect, diagnose and fix. These bugs might hide inside a code path that can only be triggered by a specific interleaving of concurrent events and only manifest under extremely rare conditions [16, 38], but the consequences can be catastrophic [1, 44].

Developers of production distributed systems use many testing techniques, such as unit testing, integration testing, stress testing, and fault injection. In spite of extensive use of these testing methods, many bugs that arise from subtle combinations of concurrency and failure events are missed during testing and get exposed only in production. However, allowing serious bugs to reach production can cost organizations a lot of money [42] and lead to customer dissatisfaction [1, 44].

We interviewed technical leaders and senior managers in Microsoft Azure regarding the top problems in distributed system development. The consensus was that one of the most critical problems today is how to improve *testing coverage* so that bugs can be uncovered *during testing* and *not in production*. The need for better testing techniques is not specific to Microsoft; other companies, such as Amazon and Google, have acknowledged [7, 39] that testing methodologies have to improve to be able to reason about the correctness of increasingly more complex distributed systems that are used in production.

Recently, the Amazon Web Services (AWS) team used formal methods "to prevent serious but subtle bugs from reaching production" [39]. The gist of their approach is to extract the high-level logic from a production system, represent this logic as specifications in the expressive TLA+ [29] language, and finally verify the specifications using a model checker. While highly effective, as demonstrated by its use in AWS, this approach falls short of "verifying that executable code correctly implements the high-level specification"' [39], and the AWS team admits that it is "not aware of any such tools that can handle distributed systems as large and complex as those being built at Amazon" [39].

We have found that checking high-level specifications is necessary but not sufficient, due to the gap between

[1]Part of the work was done while interning at Microsoft.

the specification and the executable code. Our goal is to bridge this gap. We propose a new methodology that validates high-level specifications directly on the executable code. Our methodology is different from prior approaches that required developers to either switch to an unfamiliar domain specific language [25, 10], or manually annotate and instrument their code [45]. Instead, we allow developers to systematically test *production code* by writing test harnesses in C#, a mainstream programming language. This significantly lowered the acceptance barrier for adoption by the Microsoft Azure team.

Our testing methodology is based on P# [9], an extension of C# that provides support for modeling, specification, and systematic testing of distributed systems written in the Microsoft .NET framework. To use P# for testing, the programmer has to augment the original system with three artifacts: a model of the nondeterministic execution environment of the system; a test harness that drives the system towards interesting behaviors; and safety or liveness specifications. P# then systematically exercises the test harness and validates program behaviors against the provided specifications.

The original P# paper [9] discussed language design issues and data race detection for programs written in P#, whereas this work focuses on using P# to test three distributed storage systems inside Microsoft: Azure Storage vNext; Live Table Migration; and Azure Service Fabric. We uncovered numerous bugs in these systems, including a subtle liveness bug that only intermittently manifested during stress testing for months without being fixed. Our testing approach uncovered this bug in a very small setting, which made it easy for developers to examine traces, identify, and fix the problem.

To summarize, our contributions are as follows:

- We present a new methodology for modeling, specifying properties of correctness, and systematically testing real distributed systems with P#.
- We discuss our experience of using P# to test three distributed storage systems built on top of Microsoft Azure, finding subtle bugs that could not be found with traditional testing techniques.
- We evaluate the cost and benefits of using our approach, and show that P# can detect bugs in a small setting and with easy to understand traces.

2 Testing Distributed Systems with P#

The goal of our work is to find bugs in distributed systems *before* they reach production. Typical distributed systems consist of multiple components that interact with each other via *message passing*. If messages—or unexpected failures and timeouts—are not handled properly, they can lead to subtle bugs. To expose these bugs, we use P# [9], an extension of the C# language that provides: (i) language support for *specifying* properties of *correctness*, and *modeling* the environment of distributed systems written in .NET; and (ii) a *systematic testing engine* that can explore interleavings between *distributed events*, such as the nondeterministic order of message deliveries, client requests, failures and timeouts.

Modeling using P# involves three core activities. First, the developer must modify the original system so that messages are not sent through the real network, but are instead dispatched through the `PSharp.Send(...)` method. Such modification does not need to be invasive, as it can be performed using virtual method dispatch, a C# language feature widely used for testing. Second, the developer must write a P# *test harness* that drives the system towards interesting behaviors by nondeterministically triggering various events (see §2.3). The harness is essentially a model of the environment of the system. The purpose of these first two activities is to explicitly declare all sources of nondeterminism in the system using P#. Finally, the developer must specify the criteria for correctness of an execution of the system-under-test. Specifications in P# can encode either *safety* or *liveness* [28] properties (see §2.4 and §2.5).

During testing, the P# runtime is aware of all sources of nondeterminism that were declared during modeling, and exploits this knowledge to create a scheduling point each time a nondeterministic choice has to be taken. The P# testing engine will serialize (in a single-box) the system, and repeatedly execute it from start to completion, each time exploring a potentially different set of nondeterministic choices, until it either reaches a user-supplied bound (e.g. in number of executions or time), or it hits a safety or liveness property violation. This testing process is fully automatic and has no false-positives (assuming an accurate test harness). After a bug is discovered, P# generates a trace that represents the buggy schedule, which can then be replayed to reproduce the bug. In contrast to logs typically generated during production, the P# trace provides a global order of all communication events, and thus is easier to debug.

Due to the highly asynchronous nature of distributed systems, the number of possible states that these systems can reach is exponentially large. Tools such as MoDist [48] and dBug [45] focus on testing unmodified distributed systems, but this can easily lead to state-space explosion when trying to exhaustively explore the entire state-space of a production-scale distributed storage system, such as the Azure Storage vNext. On the other hand, techniques such as TLA+ [29] have been successfully used in industry to verify specifications of complex distributed systems [39], but they are unable to verify the actual implementation.

Server

```
receive msg {
  case ClientReq:
    this.Data = message.Val;
    // Replicate data to all nodes
    foreach (sn in this.Nodes)
      sn.send(ReplReq, this.Data);
  case Sync:
    Node node = message.Id;
    Log log = message.Log;
    doSync(node, log);
}
```

```
doSync (Node sn, Log log) {
    // If the storage log is not
    // up-to-date, replicate
    if (!isUpToDate(log))
      sn.send(ReplReq, this.Data);
    else {
      this.NumReplicas++;
      if (this.NumReplicas == 3)
        this.Client.send(Ack);
    }
}
```

Storage Node

```
receive msg {
  case ReplReq:
    // Store received data
    store(message.Val);
  case Timeout:
    // Send server the log
    // upon timeout
    this.Server.send(Sync,
      this.Id, this.Log);
}
```

Client

```
while (hasNextRequest()) {
  this.Server.send(ClientReq,
    this.DataToReplicate);
  receive(Ack); // Wait for ack
}
```

Timer

```
// Send timeout to node when
// countdown reaches 0
if (this.Countdown == 0)
  this.SN.send(Timeout);
```

Figure 1: Pseudocode of a simple distributed storage system that replicates data sent by a client.

In this work, we are proposing a solution between the above two extremes: test the real implementation of one or more components against a modeled in P# environment. The benefit of our approach is that it can detect bugs in the *actual* implementation by exploring a much reduced state-space. Note that testing with P# does not come for free; developers have to invest effort and time into building a test harness using P#. However, developers already spend significant time in building test suites for distributed systems prior to deployment. The P# approach augments this effort; by investing time in modeling the environment, it offers dividends by finding more bugs (see §6). In principle, our methodology is *not* specific to P# and the .NET framework, and can be used in combination with any other programming framework that has equivalent capabilities.

2.1 The P# programming model

P# programs consist of multiple *state machines* that communicate with each other *asynchronously* by exchanging *events*. In the case of distributed systems, P# events can be used to model regular messages between system components, failures or timeouts. A P# machine declaration is similar to a C# class declaration, but a machine also contains an event queue, and one or more *states*. Each state can register *actions* to handle incoming events.

P# machines run concurrently with each other, each executing an event handling loop that dequeues the next event from the queue and handles it by invoking the registered action. An action might transition the machine to a new state, create a new machine, send an event to a machine, access a field or call a method. In P#, a send operation is *non-blocking*; the event is simply enqueued into the queue of the target machine, which will dequeue and handle the event concurrently. All this functionality is provided in a lightweight runtime library, built on top of Microsoft's Task Parallel Library [33].

2.2 An example distributed system

Figure 1 presents the pseudocode of a simple distributed storage system that was contrived for the purposes of explaining our testing methodology. The system consists of a client, a server and three storage nodes (SNs). The client sends the server a `ClientReq` message that contains data to be replicated, and then waits to get an acknowledgement before sending the next request. When the server receives `ClientReq`, it first stores the data locally (in the `Data` field), and then broadcasts a `ReplReq` message to all SNs. When an SN receives `ReplReq`, it handles the message by storing the received data locally (using the `store` method). Each SN has a timer installed, which sends periodic `Timeout` messages. Upon receiving `Timeout`, an SN sends a `Sync` message to the server that contains the storage log. The server handles `Sync` by calling the `isUpToDate` method to check if the SN log is up-to-date. If it is not, the server sends a repeat `ReplReq` message to the outdated SN. If the SN log is up-to-date, then the server increments a replica counter by one. Finally, when there are three replicas available, the server sends an `Ack` message to the client.

There are two bugs in the above example. The first bug is that the server does not keep track of unique replicas. The replica counter increments upon each up-to-date `Sync`, even if the syncing SN is already considered a replica. This means that the server might send `Ack` when fewer than three replicas exist, which is erroneous behavior. The second bug is that the server does not reset the replica counter to 0 upon sending an `Ack`. This means that when the client sends another `ClientReq`, it will never receive `Ack`, and thus block indefinitely.

2.3 Modeling the example system

To systematically test the example of Figure 1, the developer must first create a P# test harness, and then specify the correctness properties of the system. Figure 2 illustrates a test harness that can find the two bugs discussed in §2.2. Each box in the figure represents a concurrently running P# machine, while an arrow represents an event being sent from one machine to another. We use three types of boxes: (i) a box with rounded corners and thick border denotes a real component wrapped inside a P# machine; (ii) a box with thin border denotes a modeled

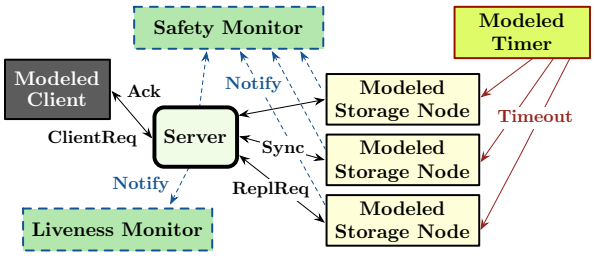

Figure 2: P# test harness for the Figure 1 example.

component in P#; and (iii) a box with dashed border denotes a special P# machine used for safety or liveness checking (see §2.4 and §2.5).

We do not model the server component since we want to test its actual implementation. The server is wrapped inside a P# machine, which is responsible for: (i) sending events via the `PSharp.Send(...)` method, instead of the real network; and (ii) delivering received events to the wrapped component. We model the SNs so that they store data in memory rather than on disk (which could be inefficient during testing). We also model the client so that it can drive the system by repeatedly sending a nondeterministically generated `ClientReq`, and then waiting for an `Ack` message. Finally, we model the timer so that P# takes control of all time-related nondeterminism in the system. This allows the P# runtime to control when a `Timeout` event will be sent to the SNs during testing, and systematically explore different schedules.

P# uses object-oriented language features such as interfaces and virtual method dispatch to connect the real code with the modeled code. Developers in industry are used to working with such features, and heavily employ them in testing production systems. In our experience, this significantly lowers the bar for engineering teams inside Microsoft to embrace P# for testing.

In §2.4 and §2.5, we discuss how safety and liveness specifications can be expressed in P# to check if the example system is correct. The details of how P# was used to model and test real distributed storage systems in Microsoft are covered in §3, §4 and §5. Interested readers can also refer to the P# GitHub repository[2] to find a manual and samples (e.g. Paxos [30] and Raft [40]).

2.4 Specifying safety properties in P#

Safety property specifications generalize the notion of source code assertions; a safety violation is a finite trace leading to an erroneous state. P# supports the usual assertions for specifying safety properties that are local to a P# machine, and also provides a way to specify global assertions by using a *safety monitor* [10], a special P# machine that can receive, but not send, events.

[2] `https://github.com/p-org/PSharp`

A safety monitor maintains local state that is modified in response to events received from ordinary (non-monitor) machines. This local state is used to maintain a history of the computation that is relevant to the property being specified. An erroneous global behavior is flagged via an assertion on the private state of the safety monitor. Thus, a monitor cleanly separates instrumentation state required for specification (inside the monitor) from program state (outside the monitor).

The first bug in the example of §2.2 is a safety bug. To find it, the developer can write a safety monitor (see Figure 2) that contains a map from unique SN ids to a Boolean value, which denotes if the SN is a replica or not. Each time an SN replicates the latest data, it notifies the monitor to update the map. Each time the server issues an `Ack`, it also notifies the monitor. If the monitor detects that an `Ack` was sent without three replicas actually existing, a safety violation is triggered.

2.5 Specifying liveness properties in P#

Liveness property specifications generalize nontermination; a liveness violation is an infinite trace that exhibits lack of progress. Typically, a liveness property is specified via a temporal logic formula [41, 29]. We take a different approach and allow the developers to write a *liveness monitor* [10]. Similar to a safety monitor, a liveness monitor can receive, but not send, events.

A liveness monitor contains two special types of states: *hot* and *cold*. A hot state denotes a point in the execution where progress is required but has not happened yet; e.g. a node has failed but a new one has not launched yet. A liveness monitor enters a hot state when it is notified that the system must make progress. The liveness monitor leaves the hot state and enters the cold state when it is notified that the system has progressed. An infinite execution is erroneous if the liveness monitor is in the hot state for an infinitely long period of time. Our liveness monitors can encode arbitrary temporal logic properties.

A liveness violation is witnessed by an *infinite* execution in which all concurrently executing P# machines are *fairly* scheduled. Since it is impossible to generate an infinite execution by executing a program for a finite amount of time, our implementation of liveness checking in P# approximates an infinite execution using several heuristics. In this work, we consider an execution longer than a large user-supplied bound as an "infinite" execution [25, 37]. Note that checking for fairness is not relevant when using this heuristic, due to our pragmatic use of a large bound.

The second bug in the example of §2.2 is a liveness bug. To detect it, the developer can write a liveness monitor (see Figure 2) that transitions from a hot state, which

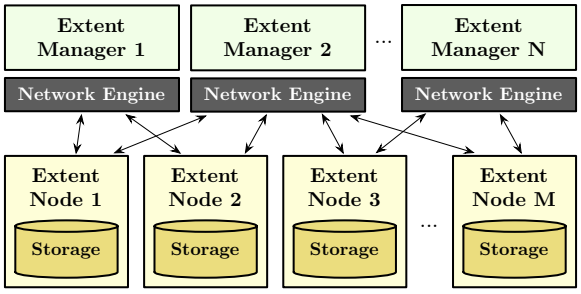

Figure 3: Top-level components for extent management in Microsoft Azure Storage vNext.

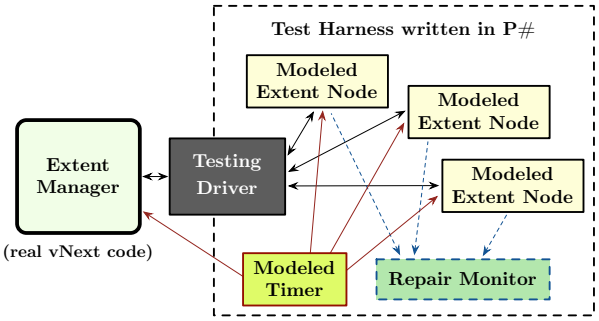

Figure 4: Real Extent Manager with its P# test harness (each box represents one P# machine).

denotes that the client sent a `ClientReq` and waits for an `Ack`, to a cold state, which denotes that the server has sent an `Ack` in response to the last `ClientReq`. Each time a server receives a `ClientReq`, it notifies the monitor to transition to the hot state. Each time the server issues an `Ack`, it notifies the monitor to transition to the cold state. If the monitor is in a hot state when the bounded infinite execution terminates, a liveness violation is triggered.

3 Case Study: Azure Storage vNext

Microsoft Azure Storage is a cloud storage system that provides customers the ability to store seemingly limitless amounts of data. It has grown from tens of petabytes in 2010 to exabytes in 2015, with the total number of objects stored exceeding 60 trillion [17].

Azure Storage vNext is the next generation storage system currently being developed for Microsoft Azure, where the primary design target is to scale the storage capacity by more than 100×. Similar to the current system, vNext employs containers, called *extents*, to store data. Extents can be several gigabytes each, consisting of many data blocks, and are replicated over multiple *Extent Nodes* (ENs). However, in contrast to the current system, which uses a Paxos-based, centralized mapping from extents to ENs [5], vNext achieves scalability by using a *distributed mapping*. In vNext, extents are divided into partitions, with each partition managed by a lightweight *Extent Manager* (ExtMgr). This partitioning is illustrated in Figure 3.

One of the responsibilities of an ExtMgr is to ensure that every managed extent maintains enough *replicas* in the system. To achieve this, an ExtMgr receives frequent periodic *heartbeat* messages from every EN that it manages. EN failure is detected by missing heartbeats. An ExtMgr also receives less frequent, but still periodic, *synchronization reports* from every EN. The sync reports list all the extents (and associated metadata) stored on the EN. Based on these two types of messages, an ExtMgr identifies which ENs have failed, and which extents are affected by the failure and are missing replicas as a result. The ExtMgr then schedules tasks to repair the affected extents and distributes the tasks to the remaining ENs. The ENs then repair the extents from the existing replicas and lazily update the ExtMgr via their next periodic sync reports. All the communications between an ExtMgr and the ENs occur via network engines installed in each component of vNext (see Figure 3).

To ensure correctness, the developers of vNext have instrumented extensive, multiple levels of testing:

1. *Unit testing*, in which emulated heartbeats and sync reports are sent to an ExtMgr. These tests check that the messages are processed as expected.

2. *Integration testing*, in which an ExtMgr is launched together with multiple ENs. An EN failure is subsequently injected. These tests check that the affected extents are eventually repaired.

3. *Stress testing*, in which an ExtMgr is launched together with multiple ENs and many extents. The test keeps repeating the following process: injects an EN failure, launches a new EN and checks that the affected extents are eventually repaired.

Despite the extensive testing efforts, the vNext developers were plagued for months by an elusive bug in the ExtMgr logic. All the unit test suites and integration test suites successfully passed on each test run. However, the stress test suite failed *from time to time* after very long executions; in these cases, certain replicas of some extents failed without subsequently being repaired.

This bug proved difficult to identify, reproduce and troubleshoot. First, an extent never being repaired is *not* a property that can be easily checked. Second, the bug appeared to manifest only in very long executions. Finally, by the time that the bug did manifest, very long execution traces had been collected, which made manual inspection tedious and ineffective.

To uncover the elusive extent repair bug in Azure Storage vNext, its developers wrote a test harness using P#. The developers expected that it was more likely for the

```
// wrapping the target vNext component in a P# machine
class ExtentManagerMachine : Machine {
  private ExtentManager ExtMgr; // real vNext code

  void Init() {
    ExtMgr = new ExtentManager();
    ExtMgr.NetEngine = new ModelNetEngine(); // model network
    ExtMgr.DisableTimer(); // disable internal timer
  }

  [OnEvent(ExtentNodeMessageEvent, DeliverMessage)]
  void DeliverMessage(ExtentNodeMessage msg) {
    // relay messages from Extent Node to Extent Manager
    ExtMgr.ProcessMessage(msg);
  }

  [OnEvent(TimerTickEvent, ProcessExtentRepair)]
  void ProcessExtentRepair() {
    // extent repair loop driven by external timer
    ExtMgr.ProcessEvent(new ExtentRepairEvent());
  }
}
```

Figure 5: The real Extent Manager is wrapped inside the `ExtentManager` P# machine.

bug to occur in the ExtMgr logic, rather than in the EN logic. Hence, they focused on testing the real ExtMgr using modeled ENs. The test harness for vNext consists of the following P# machines (as shown in Figure 4):

ExtentManager acts as a thin wrapper machine for the real ExtMgr component in vNext (see §3.1).

ExtentNode is a simple model of an EN (see §3.2).

Timer exploits the nondeterministic choice generation available in P# to model timeouts (see §3.3).

TestingDriver is responsible for driving testing scenarios, relaying messages between machines, and injecting failures (see §3.4).

RepairMonitor collects EN-related state to check if the desired liveness property is satisfied (see §3.5).

3.1 The ExtentManager machine

The real ExtMgr in vNext, which is our system-under-test, is wrapped inside the `ExtentManager` machine, as illustrated in the code snippet of Figure 5.

Real Extent Manager. The real ExtMgr (see Figure 6) contains two data structures: `ExtentCenter` and `ExtentNodeMap`. The `ExtentCenter` maps extents to their hosting ENs. It is updated upon receiving a periodic sync report from an EN. Recall that a sync report from a particular EN lists all the extents stored at the EN. Its purpose is to update the ExtMgr's possibly out-of-date view of the EN with the ground truth. The `ExtentNodeMap` maps ENs to their latest heartbeat times.

ExtMgr internally runs a periodic *EN expiration loop* that is responsible for removing ENs that have been missing heartbeats for an extended period of time, as

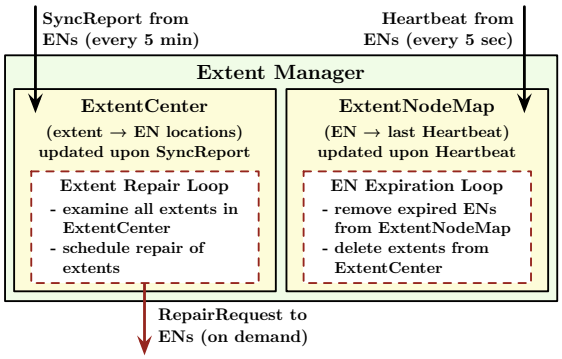

Figure 6: Internal components of the real Extent Manager in Microsoft Azure Storage vNext.

```
// network interface in vNext
class NetworkEngine {
  public virtual void SendMessage(Socket s, Message msg);
}

// modeled engine for intercepting Extent Manager messages
class ModelNetEngine : NetworkEngine {
  public override void SendMessage(Socket s, Message msg) {
    // intercept and relay Extent Manager messages
    PSharp.Send(TestingDriver, new ExtMgrMsgEvent(), s, msg);
  }
}
```

Figure 7: Modeled vNext network engine.

well as cleaning up the corresponding extent records in `ExtentCenter`. In addition, ExtMgr runs a periodic *extent repair loop* that examines all the `ExtentCenter` records, identifies extents with missing replicas, schedules extent repair tasks and sends them to the ENs.

Intercepting network messages. The real ExtMgr uses a network engine to asynchronously send messages to ENs. The P# test harness models the original network engine in vNext by overriding the original implementation. The modeled network engine (see Figure 7) intercepts all outbound messages from the ExtMgr, and invokes `PSharp.Send(...)` to asynchronously relay the messages to `TestingDriver` machine, which is responsible for dispatching the messages to the corresponding ENs. This modeled network engine replaces the real network engine in the wrapped ExtMgr (see Figure 5).

Intercepting all network messages and dispatching them through P# is important for two reasons. First, it allows P# to systematically explore the interleavings between asynchronous event handlers in the system. Second, the modeled network engine could leverage the support for controlled nondeterministic choices in P#, and choose to drop the messages in a nondeterministic fashion, in case emulating message loss is desirable (not shown in this example).

Messages coming from `ExtentNode` machines do *not* go through the modeled network engine; they are instead delivered to the `ExtentManager` machine and trigger an action that invokes the messages on the wrapped

```
// modeling Extent Node in P#
class ExtentNodeMachine : Machine {
  // leverage real vNext component whenever appropriate
  private ExtentNode.ExtentCenter ExtCtr;

  [OnEvent(ExtentCopyResponseEvent, ProcessCopyResponse)]
  void ProcessCopyResponse(ExtentCopyResponse response) {
    // extent copy response from source replica
    if (IsCopySucceeded(response)) {
      var rec = GetExtentRecord(response);
      ExtCtr.AddOrUpdate(rec); // update ExtentCenter
    }
  }

  // extent node sync logic
  [OnEvent(TimerTickEvent, ProcessExtentNodeSync)]
  void ProcessExtentNodeSync() {
    var sync = ExtCtr.GetSyncReport(); // prepare sync report
    PSharp.Send(ExtentManagerMachine,
      new ExtentNodeMessageEvent(), sync);
  }

  // extent node failure logic
  [OnEvent(FailureEvent, ProcessFailure)]
  void ProcessFailure() {
    // notifies the monitor that this EN failed
    PSharp.Notify<RepairMonitor>(new ENFailedEvent(), this);
    PSharp.Halt(); // terminate this P# machine
  }
}
```

Figure 8: Code snippet of the modeled EN.

ExtMgr with `ExtMgr.ProcessMessage` (see Figure 5). The benefit of this approach is that the real ExtMgr can be tested without modifying the original communication-related code; the ExtMgr is simply unaware of the P# test harness and behaves as if it is running in a real distributed environment and communicating with real ENs.

3.2 The ExtentNode machine

The `ExtentNode` machine is a simplified version of the original EN. The machine omits most of the complex details of a real EN, and only models the necessary logic for testing. This modeled logic includes: repairing an extent from its replica, and sending sync reports and heartbeat messages periodically to `ExtentManager`.

The P# test harness leverages components of the real vNext system whenever it is appropriate. For example, `ExtentNode` re-uses the `ExtentCenter` data structure, which is used inside a real EN for extent bookkeeping. In the modeled extent repair logic, `ExtentNode` takes action upon receiving an extent repair request from the `ExtentManager` machine. It sends a copy request to a source `ExtentNode` machine where a replica is stored. After receiving an `ExtentCopyRespEvent` event from the source machine, it updates the `ExtentCenter`, as illustrated in Figure 8.

In the modeled EN sync logic, the machine is driven by an external timer modeled in P# (see §3.3). It prepares a sync report with `extCtr.GetSyncReport(...)`, and then asynchronously sends the report to `ExtentManager` using `PSharp.Send(...)`. The `ExtentNode` machine also includes failure-related logic (see §3.4).

```
// modeling timer expiration in P#
class Timer : Machine {
  Machine Target; // target machine

  [OnEvent(RepeatedEvent, GenerateTimerTick)]
  void GenerateTimerTick() {
    // nondeterministic choice controlled by P#
    if (PSharp.Nondet())
      PSharp.Send(Target, new TimerTickEvent());
    PSharp.Send(this, new RepeatedEvent()); // loop
  }
}
```

Figure 9: Modeling timer expiration using P#.

```
// machine for driving testing scenarios in vNext
class TestingDriver : Machine {
  private HashSet<Machine> ExtentNodes; // EN machines

  void InjectNodeFailure() {
    // nondeterministically choose an EN using P#
    var node = (Machine)PSharp.Nondet(ExtentNodes);
    // fail chosen EN
    PSharp.Send(node, new FailureEvent());
  }
}
```

Figure 10: Code snippet of the TestingDriver machine.

3.3 The Timer machine

System correctness should *not* hinge on the frequency of any individual timer. Hence, it makes sense to delegate all nondeterminism due to timing-related events to P#. To achieve this, all the timers inside ExtMgr are disabled (see Figure 5), and the EN expiration loop and the extent repair loop are driven instead by timers modeled in P#, an approach also used in previous work [10].[3] Similarly, `ExtentNode` machines do *not* have internal timers either. Their periodic heartbeats and sync reports are also driven by timers modeled in P#.

Figure 9 shows the `Timer` machine in the test harness. `Timer` invokes the P# method `Nondet()`, which generates a nondeterministic choice controlled by the P# runtime. Using `Nondet()` allows the machine to nondeterministically send a timeout event to its target (the `ExtentManager` or `ExtentNode` machines). The P# testing engine has the freedom to schedule arbitrary interleavings between these timeout events and all other regular system events.

3.4 The TestingDriver machine

The `TestingDriver` machine drives two testing scenarios. In the first scenario, `TestingDriver` launches one `ExtentManager` and three `ExtentNode` machines, with a single extent on one of the ENs. It then waits for the extent to be replicated at the remaining ENs. In the second testing scenario, `TestingDriver` fails one of the

[3] We had to make a minor change to the real ExtMgr code to facilitate modeling: we added the `DisableTimer` method, which disables the real ExtMgr timer so that it can be replaced with the P# timer.

ExtentNode machines and launches a new one. It then waits for the extent to be repaired on the newly launched ExtentNode machine.

Figure 10 illustrates how TestingDriver leverages P# to inject nondeterministic failures. It uses Nondet() to nondeterministically choose an ExtentNode machine, and then sends a FailureEvent to the chosen machine to emulate an EN failure. As shown in the earlier Figure 8, the chosen ExtentNode machine processes the FailureEvent, notifies the liveness monitor of its failure (see §3.5) and terminates itself by invoking the P# method Halt(). P# not only enumerates interleavings of asynchronous event handlers, but also the values returned by calls to Nondet(), thus enumerating different failure scenarios.

3.5 The RepairMonitor liveness monitor

RepairMonitor is a P# liveness monitor (see §2.5) that transitions between a cold and a hot state. Whenever an EN fails, the monitor is notified with an ENFailedEvent event. As soon as the number of extent replicas falls below a specified target (three replicas in the current P# test harness), the monitor transitions into the hot *repairing* state, waiting for all missing replicas to be repaired. Whenever an extent replica is repaired, RepairMonitor is notified with an ExtentRepairedEvent event. When the replica number reaches again the target, the monitor transitions into the cold *repaired* state, as illustrated in the code snippet of Figure 11.

In the extent repair testing scenarios, RepairMonitor checks that it should *always eventually* end up in the cold state. Otherwise, RepairMonitor is stuck in the hot state for *infinitely* long. This indicates that the corresponding execution sequence results in an extent replica never being repaired, which is a liveness bug.

3.6 Liveness Bug in Azure Storage vNext

It took less than ten seconds for the P# testing engine to report the first occurrence of a liveness bug in vNext (see §6). Upon examining the debug trace, the developers of vNext were able to quickly confirm the bug.

The original P# trace did not include sufficient details to allow the developers to identify the root cause of the problem. Fortunately, running the test harness took very little time, so the developers were able to quickly iterate and add more refined debugging outputs in each iteration. After several iterations, the developers were able to pinpoint the exact culprit and immediately propose a solution for fixing the bug. Once the proposed solution was implemented, the developers ran the test harness again. No bugs were found during 100,000 executions, a process that only took a few minutes.

```
class RepairMonitor : Monitor {
  private HashSet<Machine> ExtentNodesWithReplica;

  // cold state: repaired
  cold state Repaired {
    [OnEvent(ENFailedEvent, ProcessENFailure)]
    void ProcessENFailure(ExtentNodeMachine en) {
        ExtentNodesWithReplica.Remove(en);
        if (ReplicaCount < Harness.REPLICA_COUNT_TARGET)
          jumpto Repairing;
    }
  }

  // hot state: repairing
  hot state Repairing {
    [OnEvent(ExtentRepairedEvent, ProcessRepairCompletion)]
    void ProcessRepairCompletion(ExtentNodeMachine en) {
      ExtentNodesWithReplica.Add(en);
      if (ReplicaCount == Harness.REPLICA_COUNT_TARGET)
        jumpto Repaired;
    }
  }
}
```

Figure 11: The RepairMonitor liveness monitor.

The liveness bug occurs in the second testing scenario, where the TestingDriver machine fails one of the ExtentNode machines and launches a new one. RepairMonitor transitions to the hot repairing state and is stuck in the state for infinitely long.

The following is one particular execution sequence resulting in this liveness bug: (i) EN_0 fails and is detected by the EN expiration loop; (ii) EN_0 is removed from ExtentNodeMap; (iii) ExtentCenter is updated and the replica count drops from 3 (which is the target) to 2; (iv) ExtMgr receives a sync report from EN_0; (v) the extent center is updated and the replica count increases again from 2 to 3. This is problematic since the replica count is equal to the target, which means that the extent repair loop will never schedule any repair task. At the same time, there are only two *true* replicas in the system, which is one less than the target. This execution sequence leads to one missing replica; repeating this process two more times would result in all replicas missing, but ExtMgr would still think that all replicas are healthy. If released to production, this bug would have caused a very serious incident of customer data unavailability.

The culprit is in step (iv), where ExtMgr receives a sync report from EN_0 after deleting the EN. This interleaving is exposed quickly by P#'s testing engine that has the control to arbitrarily interleave events. It may also occur, albeit much less frequently, during stress testing due to messages being delayed in the network. This explains why the bug only occurs from time to time during stress testing and requires long executions to manifest. In contrast, P# allows the bug to manifest quickly, the developers to iterate rapidly, the culprit to be identified promptly, and the fix to be tested effectively and thoroughly, all of which have the potential to vastly increase the productivity of distributed storage system development.

4 Case Study: Live Table Migration

Live Table Migration (MigratingTable) is a library designed to transparently migrate a key-value data set between two Microsoft Azure tables [24] (called the old table and the new table, or collectively the backend tables) while an application is accessing this data set. The MigratingTable testing effort differs from the vNext effort in two significant ways: the P# test harness was developed along with the system rather than later; and it checks complete compliance with an interface specification rather than just a single liveness property. Indeed, the P# test caught bugs throughout the development process (see x6).

During migration, each application process creates its own MigratingTable instance (MT) that refers to the same backend tables (BTs). The application performs all data access via the MT, which provides an interface named IChainTable similar to that of an Azure table (the MT assumes that the BTs provide the same interface via an adapter). A migrator job moves the data in the background. In the meantime, each logical read and write operation issued to an MT is implemented via a sequence of backend operations on the BTs according to a custom protocol. The protocol is designed to guarantee that the output of the logical operations complies with the IChainTable specification, as if all the operations were performed on a single virtual table (VT). The goal of using P# was to systematically test this property.

There are two main challenges behind testing MigratingTable: (i) the system is highly concurrent; and (ii) the logical operations accept many parameters that affect the behavior in different ways. The developers could have chosen specific input sequences, but they were not confident that these sequences would cover the combinations of parameters that might trigger bugs. Instead, they used the P#.Nondet() method to choose all of the parameters independently within certain limits. They issued the same operations to the MTs and to a reference table (RT) running a reference implementation of IChainTable specification, and compared the output. This reference implementation was reused for the BTs, since the goal was not to test the real Azure tables.

The complete test environment is shown in Figure 12. It consists of a Tables machine, which contains the BTs and RT, and serializes all operations on these tables; a set of Service machines that contain identically configured MTs; and a Migrator machine that performs the background migration. Each Service machine issues a random sequence of logical operations to its MT, which performs the backend operations on the BTs via P# events. The developers instrumented the MTs to report the linearization point of each logical operation, i.e., the time at which it takes effect on the VT, so the test harness could perform the operation on the RT at the same time. (More precisely, after processing each backend operation, the Tables machine enters a P# state that blocks all further work until the MT reports whether the backend operation was the linearization point and, if so, the logical operation has been completed on the RT. This way, the rest of the system never observes the RT to be out of sync with the VT.) For further implementation details of MigratingTable and its test harness, we refer the reader to the source code repository [36].

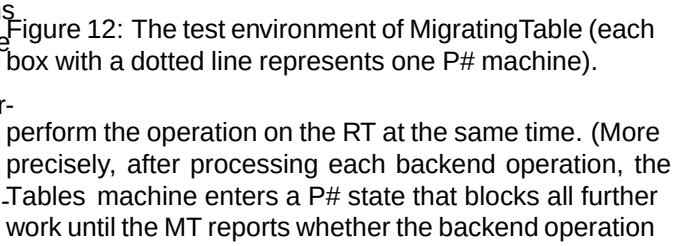

Figure 12: The test environment of MigratingTable (each box with a dotted line represents one P# machine).

5 Case Study: Azure Service Fabric

Azure Service Fabric (or Fabric for short) is a platform and API for creating reliable services that execute on a cluster of machines. The developer writes a service that receives requests (e.g. HTTP requests from a client) and mutates its state based on these requests. To make a user service reliable, Fabric launches several replicas of the service, where each replica runs as a separate process on a different node in the cluster. One replica is selected to be the primary which serves client requests; the remaining replicas are secondaries. The primary forwards state-mutating operations to the secondaries so that all replicas eventually have the same state. If the primary fails, Fabric elects one of the secondaries to be the new primary, and then launches a new secondary, which will receive a copy of the state of the new primary in order to "catch up" with the other secondaries. Fabric services are complex asynchronous and distributed applications, and are thus challenging to test.

Our primary goal was to create a P# model of Fabric (where all the Fabric asynchrony is captured and controlled by the P# runtime) to allow thorough testing of

[31] LEESATAPORNWONGSA, T., HAO, M., JOSHI, P., LUKMAN, J. F., AND GUNAWI, H. S. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), USENIX, pp. 399–414.

[32] LEESATAPORNWONGSA, T., LUKMAN, J. F., LU, S., AND GUNAWI, H. S. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ACM.

[33] LEIJEN, D., SCHULTE, W., AND BURCKHARDT, S. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (2009), ACM, pp. 227–242.

[34] LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (2010), Springer-Verlag, pp. 348–370.

[35] MADDOX, P. Testing a distributed system. *ACM Queue 13*, 7 (2015), 10–15.

[36] MCCUTCHEN, M. MigratingTable source repository. `https://github.com/mattmccutchen/MigratingTable`, 2015.

[37] MUSUVATHI, M., AND QADEER, S. Fair stateless model checking. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2008), ACM, pp. 362–371.

[38] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), USENIX, pp. 267–280.

[39] NEWCOMBE, C., RATH, T., ZHANG, F., MUNTEANU, B., BROOKER, M., AND DEARDEUFF, M. How amazon web services uses formal methods. *Communications of the ACM 58*, 4 (2015), 66–73.

[40] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference* (2014), USENIX, pp. 305–319.

[41] PNUELI, A. The temporal logic of programs. In *Proceedings of the Foundations of Computer Science* (1977), pp. 46–57.

[42] TASSEY, G. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, Planning Report 02-3* (2002).

[43] THOMSON, P., DONALDSON, A. F., AND BETTS, A. Concurrency testing using schedule bounding: An empirical study. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2014), ACM, pp. 15–28.

[44] TREYNOR, B. GoogleBlog – Today's outage for several Google services. `http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html`, 2014.

[45] ŠIMŠA, J., BRYANT, R., AND GIBSON, G. dBug: Systematic testing of unmodified distributed and multi-threaded systems. In *Proceedings of the 18th International SPIN Conference on Model Checking Software* (2011), Springer-Verlag, pp. 188–193.

[46] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), ACM, pp. 357–368.

[47] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (2009), USENIX, pp. 229–244.

[48] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (2009), USENIX, pp. 213–228.